**Qué es la programación funcional**

La programación funcional (PF) es un paradigma de programación al igual que la programación orientada a objetos (POO). La PF se basa en cálculo lambda y concretamente en composición de funciones puras para modelar las soluciones de software. En cambio, la POO está más ligada a la programación imperativa y mutable (listado de instrucciones que se van ejecutando) que tienen mucha más relación con el modelo mental de Turing que hemos comentado.

El desarrollo de software va de crear soluciones a problemas pequeños y después componerlos para solucionar un problema mayor. Es por eso que un modelo basado en funciones y en composición de las mismas como únicas herramientas para crear programas, nos brinda una forma muy elocuente de crear software.

Planteemos por ejemplo el problema de querer incrementar un numero por 1. Podemos enfocar el problema creando una función que resuelva directamente el problema:

const inc = x => x + 1;

O podemos pensar en solucionar primero el problema de sumar dos números y después crear nuestra función **inc** mediante una composición:

const add = x => y => x + y;

const inc = add(1);

Nuestra función **add** toma un valor X y devuelve una función que toma otro valor Y. Finalmente devuelve la suma de los dos números.

La función **inc** es solo una composición (en este caso usando aplicación parcial) de **add**.

Si mañana tenemos que crear más funciones ‘inc’, simplemente tendremos que seguir especlializando a la función **add**:

const inc2 = add(2);

const inc3 = add(3);

Además, podemos crear funciones completamente nuevas componiendo varias ya existentes:

const head = arr => arr[0];

const splitBySpace = str => str.split(' ');

const firstWord = compose(head, splitBySpace);

const toUpperCase = str => str.toUpperCase();

const toUpperCaseFirstWord = compose(toUpperCase, firstWord);

toUpperCaseFirstWord('Hello World') *// HELLO*

Hemos definido una serie de funciones y después las hemos ido componiendo para crear funciones nuevas.

Cuando hablamos de composición nos referimos simplemente a usar el resultado de una función como ‘input’ de otra. Siempre que los tipos coincidan (es decir, mi función A devuelve el tipo X y mi función B recibe el tipo X) podremos componerlas.

**Características de la programación funcional**

A lo largo de las implementaciones de los modelos anteriormente comentados, cada paradigma fue definiendo una serie de peculiaridades. Así las características que definen al paradigma funcional hoy por hoy son las siguientes:

1. No hay estado global.
2. Todas las funciones son puras: Dado un mismo input siempre devolvemos el mismo output.
3. Todos los valores son inmutables: Lo único que podemos hacer es generar nuevos valores.
4. No hay bucles: La iteración se realiza usando recursividad.

Como el modelo de cálculo lambda carecía de “cinta” para conservar el estado del programa, este se tenía que ir regenerando a través de la composición de funciones y la recursividad.

Un buen ejemplo de cómo podemos crear cualquier cosa usando funciones es la lógica combinatoria, una variante del cálculo lambda que provee un set limitado de funciones combinadoras:

*// funciones combinadoras*

const I = x => x;

const K = x => y => x;

const V = x => y => z => z(x)(y);

*// implementación de una tupla*

const first = I;

const second = K(I);

const tuple = V;

const myTuple = tuple('Hello')('World');

myTuple(first); *// 'Hello'*

myTuple(second); *// 'World'*

Cumpliendo las características arriba comentadas hemos implementado una tupla básica usando solo funciones.

Los combinadores son excelentes ejemplos de programación funcional y están presentes hoy en día más de lo que creemos. Por ejemplo, la famosa aceleradora de startups de capital riesgo estadounidense [Y combinator](https://es.wikipedia.org/wiki/Y_Combinator), toma el nombre el combinador ‘Y’ de lógica combinatoria.

Además, según la implementación, el paradigma de programación funcional también se suele asociar a lo que en teoría de programación se conoce como “lazy evaluation” o evaluación perezosa. Esta estrategia de evaluación que implementan lenguajes como Haskell consiste en no evaluar ninguna expresión hasta que el valor se necesite realmente. Así podemos definir estructuras de datos infinitas que por lo general hacen más sencilla la implementación de ciertos algoritmos.

Por ejemplo, si quisiéramos representar en una lista todo el conjunto de los números naturales usando Haskell, podríamos hacer esto:

n = [0..] -- lista infinita representando todos los numeros naturales

doubles = map (\*2) n -- doblamos todos los numeros naturales

Al ser evaluado perezosamente, haskell no generará la lista hasta que realmente hagamos algo con ella:

main = print (take 10 doubles) -- [0,2,4,6,8,10,12,14,16,18]

En el ejemplo anterior estamos cogiendo los 10 primeros números de un conjunto infinito e imprimiéndolos por pantalla. Esto es posible porque los 10 primeros números naturales duplicados son una determinación en contraposición con la lista del doble de todos los números naturales que sería una indeterminación, representable para pensar en abstracto, pero no evaluable (obviamente).